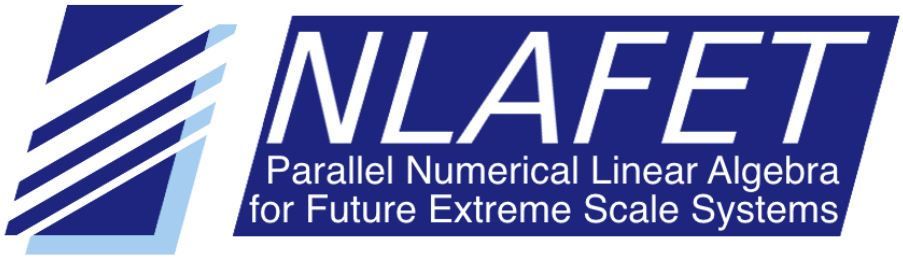


H2020–FETHPC–2014: GA 671633

D2.8

Prototypes for two-sided bidiagonal factorization

April 2017

Document information

Scheduled delivery	2017-04-30
Actual delivery	(not yet delivered)
Version	0.1
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2017-01-18	Mawussi Zounon	Draft	0.1	Initial version of document produced.

Internal reviewers

First Last (UMU/UNIMAN/STFC/INRIA)
First Last (UMU/UNIMAN/STFC/INRIA)

Author(s)

- Negin Bagherpour, The University of Manchester, Manchester, UK.
- Jack Dongarra. Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA; Oak Ridge National Laboratory, TN, USA; The University of Manchester, Manchester, UK.
- Sven Hammarling The University of Manchester, Manchester, UK.
- Nicholas J. Higham, The University of Manchester, Manchester, UK.
- Samuel D. Relton, The University of Manchester, Manchester, UK.
- Mawussi Zounon, The University of Manchester, Manchester, UK.

Internal reviewers

Reviwer 1 and Reviwer 2, Affiliation.

Contributors

- The Innovative Computing Laboratory (ICL), the University of Tennessee.
- Intel Corporation.

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633. This material is based upon work supported in part by the National Science Foundation under Grants No. CSR 1514286 and ACI-1339822, NVIDIA, the Department of Energy.

Table of Contents

List of Figures

Abstract

We present different algorithms for computing a two-sided bidiagonal factorization of a matrix. This factorization is required in many scientific applications, since a bidiagonal form is a stepping stone on the path towards the singular value decomposition (SVD). To this end, we split the bidiagonalization into a two-stage process. During the first stage, the full matrix is reduced to band bidiagonal form: this stage consists of compute-intensive operations and represents the dominant cost (in terms of total flops) to solution. On the other hand, the second stage is memory-bound and consists of reducing the band bidiagonal matrix to bidiagonal form. In the context of tile algorithms and task-based programming, we focus primarily on the first stage and propose two different implementations. We analyse the pros and cons of each of these two algorithms and discuss how a careful data dependency analysis can help remove unnecessary task dependencies and improve the performance further. Through experimental results on both a 20-core Intel Xeon multicore node and a 68-core Intel Xeon Phi KNL, we demonstrate that our resulting task-based OpenMP prototype is competitive with state-of-the-art implementations. Finally, we discuss potential solutions for the design of the second stage which is currently in progress.

1 Introduction

The main objective of this work is to investigate new strategies for two-sided bidiagonal factorization, which is widely used to transform a full matrix into bidiagonal form using orthogonal transformations. A matrix bidiagonalization is required as a precurser to computing the singular value decomposition (SVD). The SVD of an $m \times n$ matrix A is given by: $A = U\Sigma V^T (A = U\Sigma V^H$ if A is complex) where U and V are orthogonal (unitary) and Σ is an $m \times n$ matrix with real diagonal elements, σ_i , commonly ordered such that: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{min(m,n)} \geq 0$. The σ_i are known as the singular values of A and the first $\min(m, n)$ columns of U and V are the left and right singular vectors of A, respectively.

For a given $m \times n$ matrix A, the traditional approach, for computing the singular values and optionally the singular vectors, consists of the following steps:

1. Reduction of the matrix A to a bidiagonal form: $A = U_1 B V_1^T$ if A is real ($A = U_1 B V_1^H$ if A is complex), where U_1 and V_1 are orthogonal (unitary if A is complex), and B is real and upper bidiagonal when $m \geq n$ or lower bidiagonal when $m < n$, so that B is nonzero on only the main diagonal and either the first superdiagonal (if $m \geq n$) or the first subdiagonal (if$m < n$).

2. The SVD computation of the bidiagonal matrix B: $B = U_2 \Sigma V_2^T$, where U_2 and V_2 are orthogonal, and Σ is diagonal with entries consisting of the singular values of A.

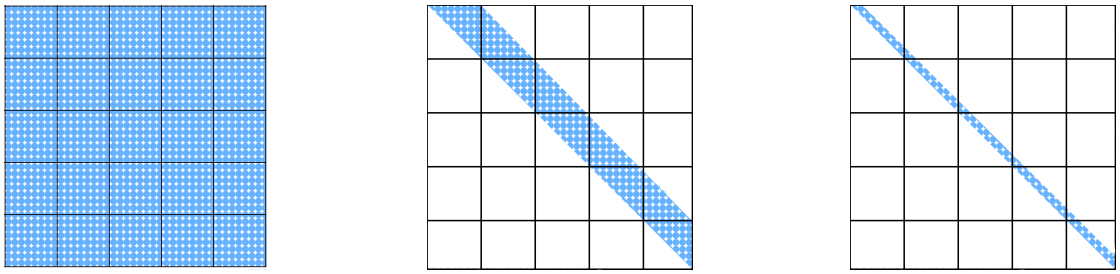
3. If required, the singular vectors of A are computed as follows $U = U_1 U_2$ and $V = V_1 V_2$.

The SVD decomposition algorithm described above was introduced in 1965 by Golub and Kahan [1]. While the last two steps have been considerably optimized for modern architectures, the first step: the reduction to bidiagonal form remains limited by memory-bound operations. In fact, in the algorithm proposed by Golub, the reduction to bidiagonal form is achieved by applying a QR step on the first column, followed by a

LQ step on the first row, then a QR step on the second column, and so on. Since this algorithm processes one column (row) at a time thanks to Householder transformation, it is limited to Level-2 BLAS kernels (and is hence memory-bound). In 1989, to improve the bidiagonalization step, Dongarra, Sorensen, and Hammarling [2] proposed a new variant which has the advantage of exploiting Level-3 BLAS operations. Instead of applying the householder transformations to one column at a time, this algorithm uses a aggregated householder transformations strategy [3] to process a few columns at a time, giving us the opportunity to use compute-intensive Level-3 BLAS kernels. This modification to the bidiagonalization algorithm helps to reformulate approximately 50% of the process as Level-3 BLAS operations as reported by Großer and Lang in [4].

To address the remaining 50% of Level-2 BLAS operations in the algorithm, in 1999, Großer and Lang introduced a two-staged approach. As illustrated in Figure 1, the first stage consists of reducing the full matrix into a band bidiagonal form (Figure 1b) using only compute-intensive kernels; and a second stage to reduce the band matrix to a bidiagonal form (Figure 1c) with memory-bound kernels. Since the first stage is a compute-intensive algorithm and represents the dominant part of the process, it significantly increases the overall performance of the bidiagonalization process. This two-stage algorithm may introduce extra flops but as reported by Azzam et al. [5] it is still efficient in terms of time to solution.

In this work, we consider the two-stage bidiagonalization algorithm with a special focus on the first stage. We compare various design options making use of tile algorithms and the task-based programming model. As a result, we propose a prototype based on the OpenMP 4.0 runtime system which is competitive with state-of-the-art implementations. The remainder of this paper is organized as follows. Section 2 explains the background to tiled algorithms. In section 3 we discuss and analyze different strategies for implementing the reduction from a full matrix to band bidiagonal form, followed by performance analyses and a discussion of the potential performance improvement. Section 4 investigates different solutions for transforming the band bidiagonal matrix to bidiagonal form before providing some concluding remarks in Section 5.



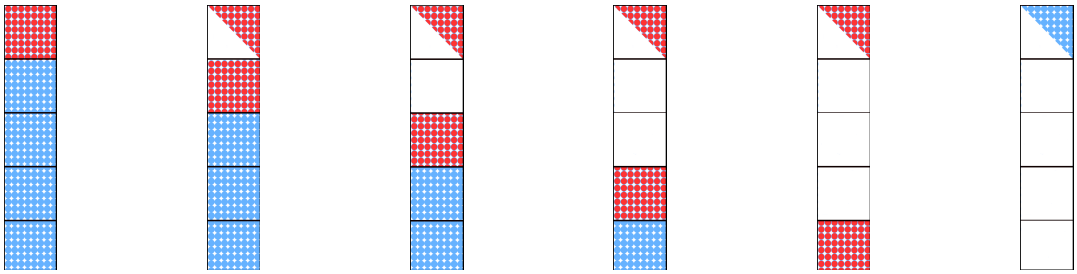
(a) Full 5×5 tile matrix. (b) Band bidiagonal form. (c) Bidiagonal form.

Figure 1: Illustration of the two-stage bidiagonal reduction process.

2 Principles of tile algorithms

In order to use modern many/multi-core shared memory architectures at full efficiency, many LAPACK library algorithms initially powered by block algorithms have been redesigned into tile algorithms. This led to a new generation of linear algebra libraries such as PLASMA [6] and FLAME [7]. The key idea is that, instead of operating on block-

columns as found in LAPACK, tile algorithms operates at a finer granularity by dividing the whole matrix into small square tiles which are more likely to fit into the L2 cache of a CPU. As illustrated in Figure 1a, the original matrix has been converted into a 5-by-5 tile matrix. One of the advantages of working at tile granularity is that it provides more room for parallelism with many tasks to keep all computational cores busy. Another advantage of tile algorithms is that they alleviate the fork-join overhead inherent to parallelized LAPACK block algorithms [8]. In fact, the order of execution of the tasks in tile algorithms are commonly represented in form of a Directed Acyclic Graph (DAG) where each node represents a task, while the edges represent the data dependencies between the tasks. These tasks are then scheduled thanks to a runtime system which check the dependencies and takes care of launching tasks on appropriate cores.



(a) QR (b) TSQRT (c) TSQRT (d) TSQRT (e) TSQRT (f) End

Figure 2: Panel factorization using the triangle on top of square QR factorization kernel (TSQRT).

For the sake of illustration we show in Figure 2 the tile algorithm variant of QR factorization for a panel (block-column). Following the example shown in Figure 1, the panel has been divided into 5 tiles. This algorithm involves two tile kernels: a standard QR factorization kernel to factorize the top-most tile (Figure 2a), and a specialized kernel to use the top triangular matrix from the first QR factorization to zero the square tiles below (Figures 2b-2e). This kernel is denoted TSQRT and stands for "triangle on top of square tile QR factorization". In the next few sections we will introduce other specialized kernels, designed to operate on tiles.

3 Reduction to band bidiagonal form

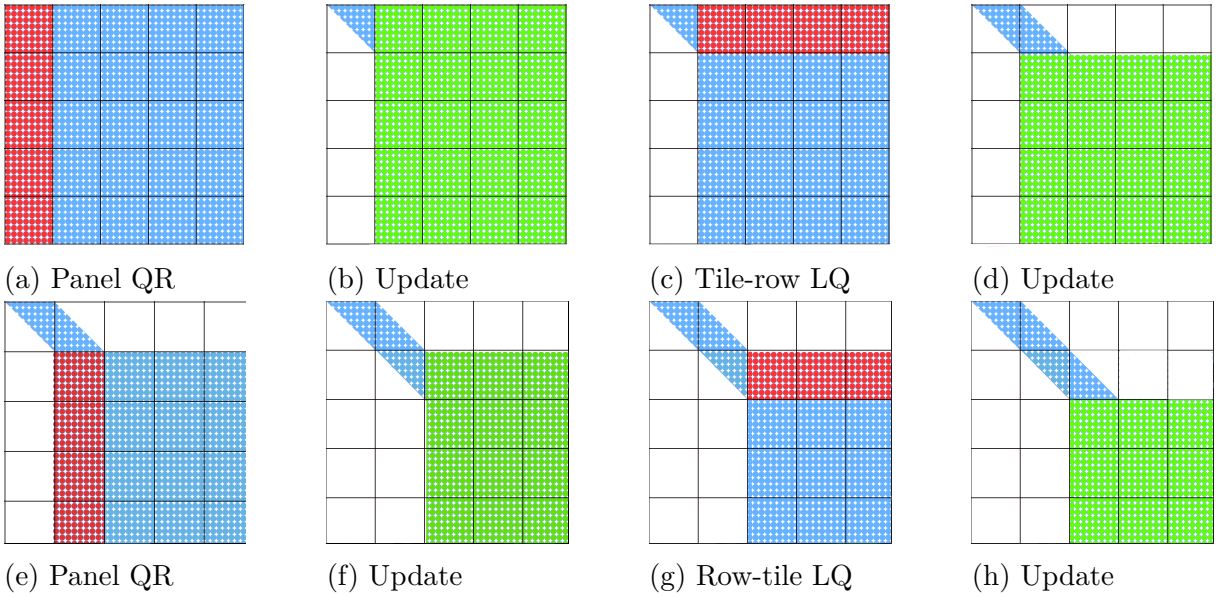
The idea of using a tile algorithm to factorize a full matrix into band bidiagonal is not new. The approach was first introduced in 2010 by Dongarra et al. [9], and improved in 2013 [5] by the same authors. This section aims at evaluating different options for designing such an algorithm before moving to the second stage of the factorization.

We identified two main design options: (1) a "late update" strategy, and (2) an "early update" strategy. Each of these strategies are introduced and assessed in the subsections below.

3.1 Late update strategy

To reduce a full $mt \times nt$ tile matrix A to a band bidiagonal form, the late update strategy begins by applying a QR factorization to the first panel $A(1 : mt, 1)$ (using MATLAB

indexing) (see Figure 3a), then updating the trailing submatrix $A(1 : mt, 2 : nt)$ (Figure 3b), followed by an LQ factorization of the first tile-row, ignoring the first column (Figure 3c), and the update of the corresponding trailing matrix (Figure 3d). This procedure is repeated until the whole matrix is reduced to band bidiagonal form, as illustrated in Figure 3. It is important to notice that we describe this strategy at tile-column (panel) and tile-row granularity but tile algorithms are used underneath to process each panel and the update of the trailing matrix. We can think of this as a direct translation of the LAPACK column-oriented procedure into a more modern tile-based algorithm.



(a) Panel QR (b) Update (c) Tile-row LQ (d) Update

(e) Panel QR (f) Update (g) Row-tile LQ (h) Update

Figure 3: Reduction from general matrix to band bidiagonal form using the late update strategy.

The main advantage of this strategy is its simplicity, the algorithm is very close to the LAPACK version. However the individual steps of the QR factorization illustrated in Figure 2, shows that a full factorization of a panel starts with the QR factorization of the first tile (Figure 2a), which means only one CPU core is busy while the others remain idle. Once the first tile is factorized the rest of work (to annihilate the tiles below using the TSQRT kernel) can commence. Since the elimination procedure modifies the entries of the top triangular tile only one square tile can be eliminated at a time to guarantee a correct result. Put differently, the panel factorization introduces synchronization points in the algorithm that may lead to performance penalties. The sequential nature of the panel factorization can be clearly observed in Figure 4, which shows the DAG of the corresponding operations.

One can overcome this synchronization issue by overlapping the panel factorization and the update steps. The algorithm can then progress onto the next step even though the previous step is not completed, as long as we respect the data dependencies. In addition, when working on a panel we can take advantage of "tall and skinny" QR factorization strategies: providing elegant methods to introduce parallelism during a panel factorization. This leads us to the early update strategy.

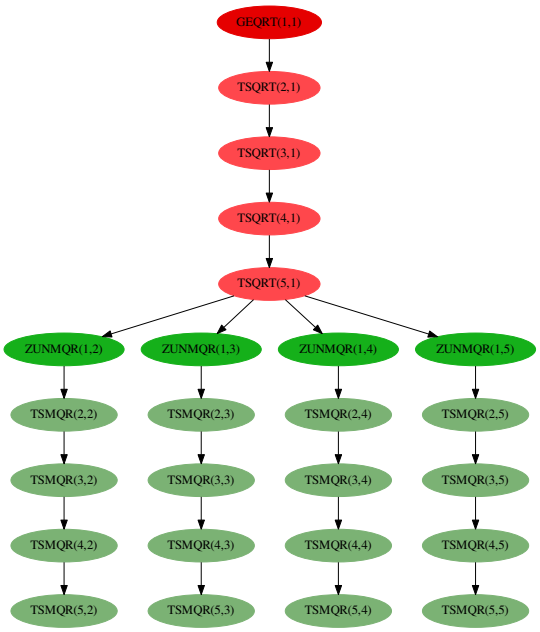


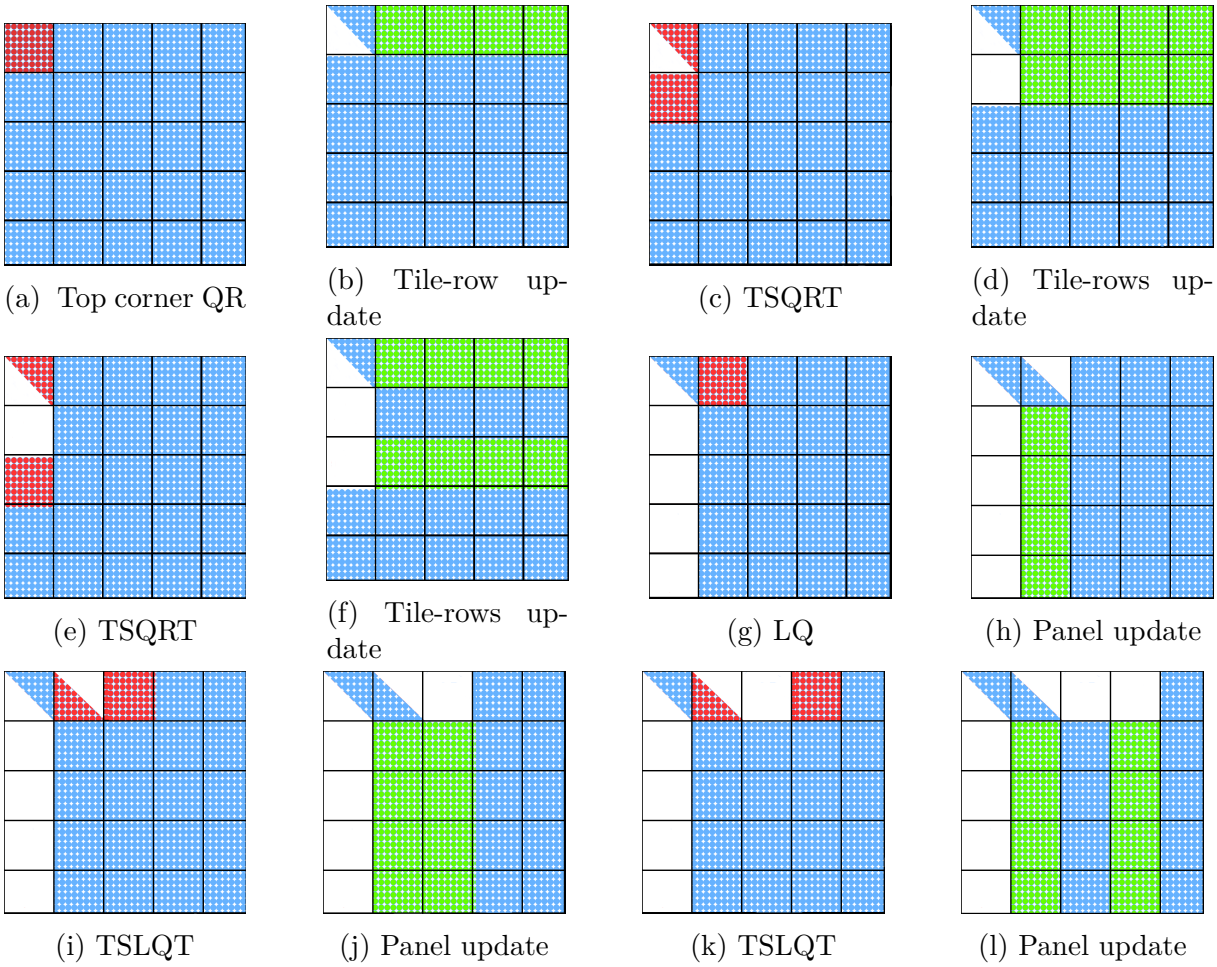
Figure 4: DAG of the late update strategy for band reduction: this partial view is limited to the factorization of the first panel and the update of the corresponding trailing matrix.

3.2 Early update strategy

To reduce a full tile matrix to a band bidiagonal form, the early update strategy also completes the QR factorization of the first panel and the update before starting the LQ process, but the operations are launched in a completely different order. Instead of completing the full factorization of a panel before applying the updates as before, the early update strategy looks down the panel, factorizes each tile, and then launches its corresponding updates immediately.

As depicted in Figure 5, the algorithm starts with the factorization of the tile $A(1,1)$ (Figure 5a), before applying the corresponding transformations to the rest of the tile-row (Figure 5b). Once the first tile is factorized the remaining tiles in the panel are successively eliminated and, in the same way, the trailing submatrices are updated immediately (as illustrated in Figures 5c and 5d as well as in Figures 5e and 5f). In addition, the updates in Figure 5d can be computed in parallel with the next tile elimination in Figure 5e. The same approach is used during the LQ factorization step.

As illustrated in the DAG (Figure 6), the early update approach considerably reduces the severity of the synchronization points. However, all the ZUNMQR kernels in charge of updating the first tile-row have to be completed before starting any TSQRT kernel, since the ZUNMQR kernels use the entries of the top left tile and those entries will be overwritten if any TSQRT kernels are launched before the end of the first tile-row update. This explains the bottleneck at TSQRT(2,1) in the DAG. But the completion of the first TSQRT kernel unlocks many tasks and leads to high levels of parallelism.



(a) Top corner QR (b) Tile-row update (c) TSQRT (d) Tile-rows update

(e) TSQRT (f) Tile-rows update (g) LQ (h) Panel update

(i) TSLQT (j) Panel update (k) TSLQT (l) Panel update

Figure 5: Reduction from general matrix to band bidiagonal form using early update strategy.

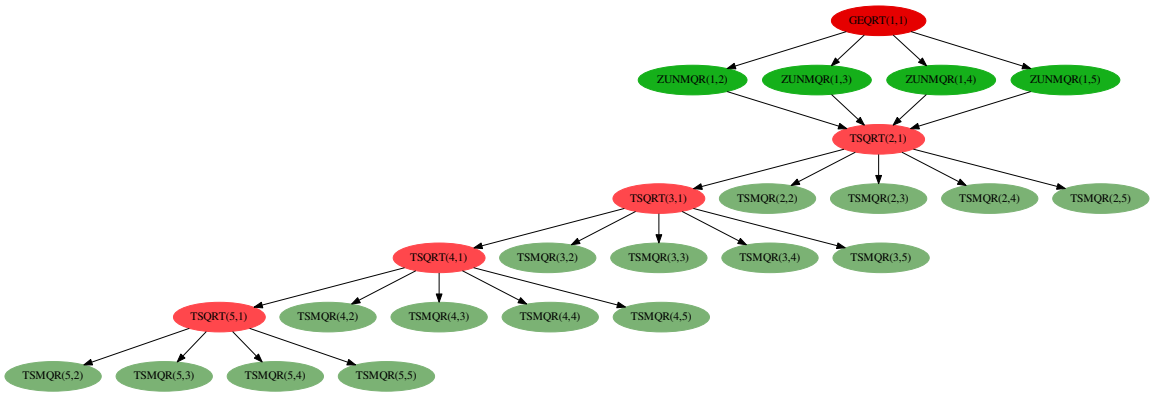
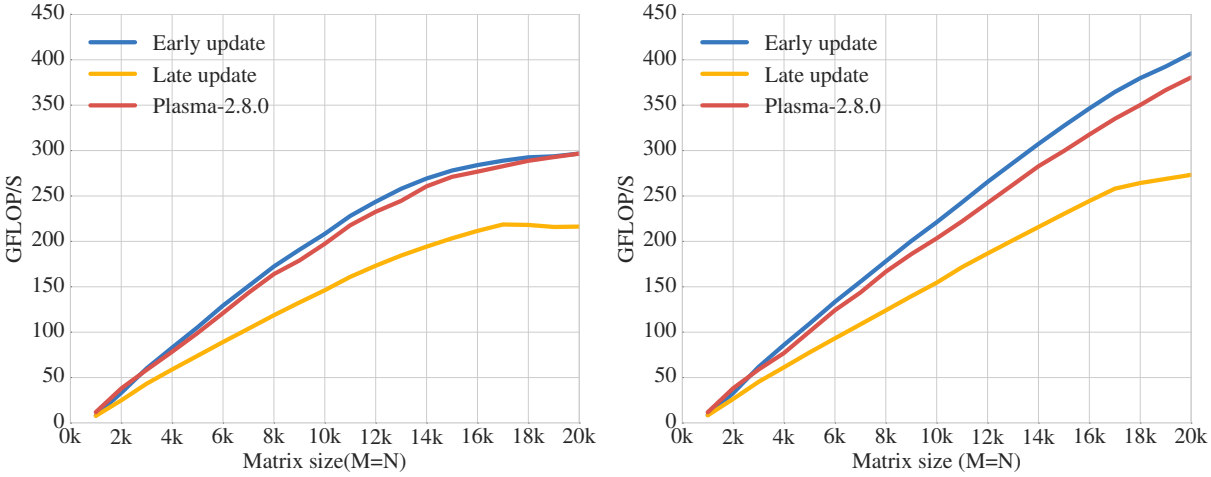


Figure 6: DAG for the early update strategy for band reduction: this partial view is limited to the factorization of the first panel and the update of the corresponding trailing matrix.

3.3 Experimental results

In order to assess the performance of each strategy, we design an OpenMP task-based version of each of the two strategies. In all the experiments within this section, `Early update` and `Late update` will denote the two strategies explained above, respectively. We also compare these strategies to the PLASMA 2.8.0 kernel designed for reduction of a full matrix to band bidiagonal form. The experiments have been performed with a NUMA node (two-socket Xeon(R) CPU E5-2650 v3 @ 2.30GHz–Haswell) and a 68-core Intel KNL[1] and all the computations are done in double precision arithmetic. The performance (Gflop/s) displayed is calculated by dividing the standard theoretical flops by the time to solution.



(a) Results with DDR4 (b) Results with MCDRAM

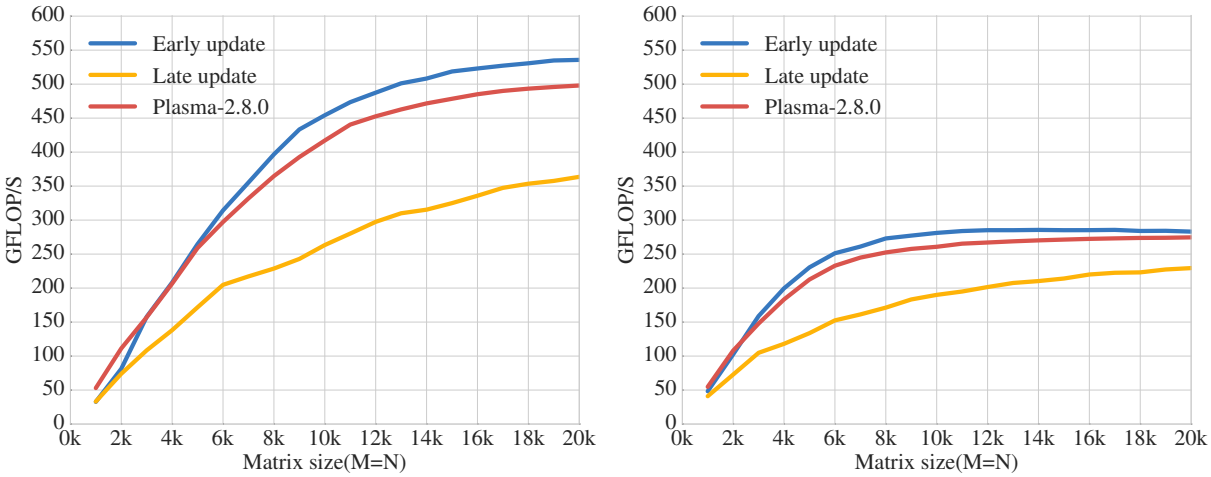
Figure 7: Performance comparison of different implementations of DGE2GB using 68 threads on the Intel KNL with square matrices ranging in size from $1,000 \times 1,000$ to $20,000 \times 20,000$.

In addition to the traditional DDR4, the Intel KNL has a high bandwidth memory called Multi-Channel DRAM (MCDRAM) with a bandwidth four times greater than the DDR4 bandwidth, but with a storage capacity limited to 16GB. There are different configuration options of MCDRAM, but in this experiment it has been configured in flat mode i.e. the 16GB memory can be directly allocated from within an application as is the case for DDR4. In our experiments we provide results for both DDR4 and MCDRAM.

As illustrated in Figure 7a where the data is allocated in DDR4, the early update approach gives the best performance with a slight advantage over the Plasma-2.8.0 implementation. The significant gap between the early update and late update strategies is consistent with our expectations and confirms the benefit of the early update strategy. In Figure 7b where data is allocated in the MCDRAM instead of the regular DDR4, the difference between the strategies is fairly similar, although all implementations benefit from the increased memory access speed.

We obtained a similar result on a NUMA node (2x Intel Xeon(R) CPU E5-2650 v3) using 20 threads and 10 threads in Figure 11a and Figure 11a respectively. However, unlike for the 68-core Intel KNL, the late update strategy exhibited a more severe performance penalty.

[1] https://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core



(a) Results with 20 threads

(b) Results with 10 threads

Figure 8: Performance comparison of different implementations of DGE2GB on a 2x Intel Xeon(R) CPU E5-2650 v3 @ 2.30GHz (20 cores), with square matrices ranging in size from $1,000 \times 1,000$ to $20,000 \times 20,000$. The experiment with 20 threads is performed with the NUMA configuration "numactl –interleave=all" while the experiment with 10 threads using only one 10-core socket.

In the next subsection we investigate ways to remove the bottlenecks in both synchronisation points in both the early and late update strategies.

3.4 Potential improvements

One way to solve the performance issue discussed for both the late update and the early update strategy is to revisit the data dependencies and remove all unnecessary synchronizations. In Figure 3 for example, since the algorithm is task based, a part of the trailing matrix update in Figure 3b could start before the completion of the panel factorization (Figure 3a), if its dependencies are satisfied. In the same way, once the first tile-row is updated, the update of the other tile-rows below could follow as soon as their corresponding tile in the panel is eliminated. Unfortunately, all these parallelisms are not exploited in the version presented above.

In fact, the update of the first tile-row uses the top left corner tile $A(1,1)$ as input (read dependency), while the TSQRT kernel in charge of the elimination of the square tiles in the panel modifies $A(1,1)$ (read and write dependency). The update kernel then waits until the completion of the panel factorization. But a further analysis of the algorithm helps to realize that as illustrated in Figure 2, once $A(1,1)$ is factorized, the elimination operations applied to the rest of the panel modify only the upper triangular part of $A(1,1)$. On the other hand, the update of the first tile-row requires only reflectors stored in the lower triangular part of $A(1,1)$. Therefore the panel factorization and the update of the first tile-row could be done in parallel.

Currently our OpenMP implementation supports dependencies only between full tiles. This introduces an unnecessary dependency between the first tile-row update and the first tile-column elimination through the A(1,1) tile. To solve this we might want to split all A(i,i) tiles into upper and lower parts.

Alternatively, one can modify the way OpenMP expresses its data dependencies. To

illustrate this, lets consider an $nb \times nb$ tile B. The standard way to express a read or input dependency on B is

```
#pragma omp task depend(in:B[0:nb*nb])
```

while write or output dependency on B would be

```
#pragma omp task depend(out:B[0:nb*nb]).
```

The OpenMP runtime system then ensures that the dependency criteria are satisfied on each of the $nb \times nb$ entries before executing the corresponding kernel. But instead of using `depend(in:B[0:nb*nb])`, if one uses `depend(in:B[i])` the runtime system will check only the i^{th} entry of the tile to decide whether to execute the kernel or not. Although specifying the whole ranges is the recommended way, using only a single entry is an alternative in cases where only some entries of the tile are used and the i^{th} entry is representative of their dependency. Since the tile entries are stored in column major format, an input and output dependency on the upper triangular tile can be simulated by `#pragma omp task depend(inout:B[0])` (zero is the starting index of the upper triangular tile with respect to C language and column major storing) and an input dependency on the lower triangular by `#pragma omp task depend(in:B[1])` (one is the starting index of the lower triangular tile).

Expressing A(i,i) tiles dependency at the upper/lower triangular granularity helps us achieving a highly parallel kernel as demonstrated by the corresponding DAG depicted in Figure 9.

We also applied this modification to the early update strategy to remove the synchronisation point observed previously. Infact, since we are working at the triangular tile granularity this releases all unnecessarily dependencies, the panel and tile strategies should provide comparative results.

To assess the effectiveness of this improvement, We reproduced the same experiments but now with some critical dependencies expressed at triangular tile granularity.

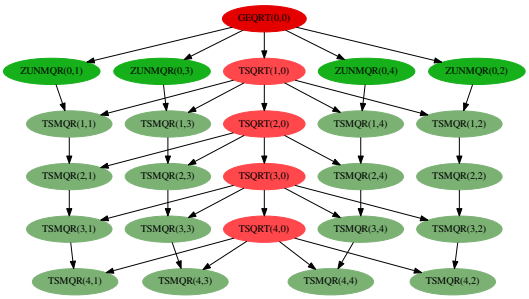
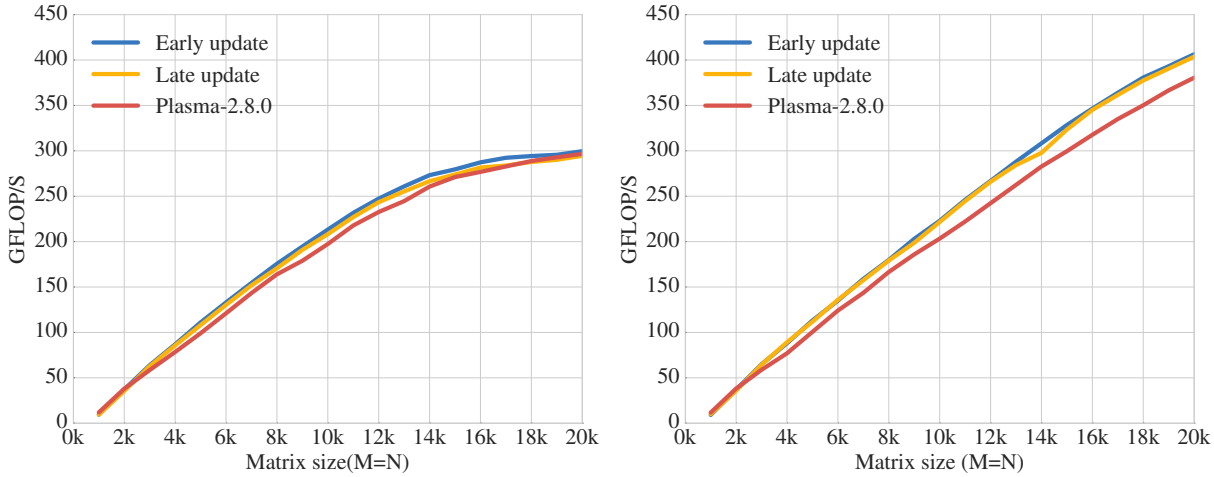


Figure 9: DAG for the late update strategy for band reduction when A(i,i) dependencies are expressed at the upper/lower tile granularity. This partial view is limited to the factorization of the first panel and the update of the corresponding trailing matrix.

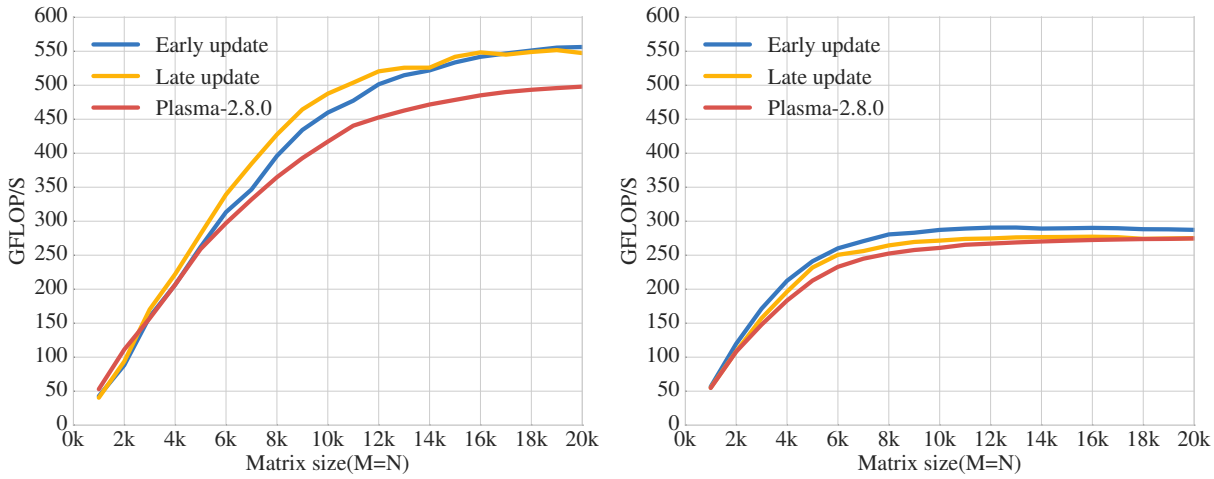
The effectiveness of the modification is illustrated by the improvement of the Intel 68-core KNL performance results in Figure 10 where the early update and the late update strategies almost overlap and both are slightly better than Plasma-2.8.0. This observation is consistent with results with DDR4 as well as for results with MCDRAM. There is also a slight improvement in the performance achieved by the early update strategy compared to the results in Figure 7 where all dependencies were expressed at full tile granularity. We have also observed similar results for the experiments illustrated in Figure 11 with the Intel Haswell NUMA node.

As reported in [5], the SVD implementation based on the Plasma-2.8.0 bidiagonalization kernel could be two times faster than Intel's Math Kernel Library (MKL), when all



(a) Results with DDR4 (b) Results with MCDRAM

Figure 10: Performance comparison of different implementations of DGE2GB using 68 threads on the Intel KNL with different square matrices ranging in size from $1,000 \times 1,000$ to $20,000 \times 20,000$. The code has been modified to express some data dependencies at `upper/lower triangular tile` granularity.



(a) Results with 20 threads (b) Results with 10 threads

Figure 11: Performance comparison of different implementations of DGE2GB on a NUMA node with 2x Intel Xeon(R) CPU E5-2650 v3 @ 2.30GHz (20 cores), with square matrices ranging in size from $1,000 \times 1,000$ to $20,000 \times 20,000$. The experiment with 20 threads is performed with the NUMA configuration "numactl –interleave=all" while the experiment with 10 threads used only one 10-core socket. The code has been modified to express some data dependencies at `upper/lower triangular tile` granularity.

the singular vectors are requested and up to 10 times faster if only the singular values are required. This shows the efficiency of the Plasma-2.8.0 bidiagonalization kernel, and the performance of our OpenMP implementation compared Plasma-2.8.0 demonstrates the effectiveness of our prototype.

Furthermore, the hint of working at triangular tile granularity helps to gain only a slight performance improvement for the early update strategy while it requires declaring OpenMP dependencies in a non-conventional way. For this reason, it makes sense to keep

the tile oriented strategy with dependencies at full tile granularity, as it is reasonably efficient and respects the standard OpenMP programming conventions. In addition by declaring data dependencies in a conventional way the code can be safely extended to a distributed memory environment.

4 Band reduction to bidiagonal form

In this section we briefly discuss the second stage of the reduction to bidiagonal form. In fact the standard bidiagonalization algorithm as proposed by Golub and Kahan requires $\frac{8}{3}n^3$ flops to reduce an $n \times n$ full matrix to a bidiagonal form. Similarly, the reduction to block bidiagonal form with a block size of nb (using $nb \times nb$ tiles) performs $\frac{8}{3}n \times (n - nb)^2$ flops while the second stage (moving from block bidiagonal to bidiagonal) requires only $4n^2 \times nb$ flops [10]. Since nb is very small compared to n, the first stage is the dominant part of the algorithm. However since the second stage is memory-bound it can penalize the overall performance if is not implemented efficiently.

We identify two potential solutions for the second stage. The first consists of using an existing LAPACK routine: while LAPACK does not provide any routine to reduce a full matrix to band bidiagonal form, the GBBRD routine available in LAPACK is designed to reduce a band matrix to bidiagonal form. Relying on a vendor optimized GEBBRD kernel (for instance Intel MKL) this can be a reasonable solution.

On the other hand, we can implement our own state-of-the-art solution for the second stage. In particular we can revisit the cache-aware and task coalescing techniques introduced by Haidar et al. in [11] which seems to have a good potential for high performance. We are currently working on an implementation of this algorithm in OpenMP, after which we can assess its efficiency by comparing against the optimized MKL GBBRD kernel.

5 Concluding remarks

In this paper we have studied different algorithms for the two-stage bidiagonalization with a special focus on the first stage: reduction from a full matrix to a band bidiagonal form. We proved that even though the late update strategy is simple to implement and follows the LAPACK-style, it suffers from numerous synchronisation points that could be released by the early update approach.

We also discuss potential improvements by relaxing some OpenMP standard dependency expressing rules. We demonstrated that while the improvement makes sense for the late update approach, the gain is not significant enough for the early update strategy to sacrifice the robustness of the kernel. Furthermore we have demonstrated the efficiency of our implementation by showing that it is competitive with the state-of-the-art kernels.

For the second stage: reduction of band bidiagonal matrix to bidiagonal form, we have identified two potential solutions which are currently in development.

Since this work focused on square matrices, future work will be devoted to designing specialized kernels for very tall matrices (number of lines more than 3x the number of columns) which require different algorithms to keep the computational resources working at full efficiency. Also the current version of our prototype is based on OpenMP runtime system which is limited to shared memory systems. In the future, we will consider using distributed memory runtime systems such as StarPU [12] and PaRSEC [13] to achieve an extreme-scale implementation.

References

[1] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[2] Jack J Dongarra, Danny C Sorensen, and Sven J Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, 1989.

[3] Christian Bischof and Charles Van Loan. The wy representation for products of householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

[4] Benedikt Großer and Bruno Lang. Efficient parallel reduction to bidiagonal form. *Parallel Computing*, 25(8):969–986, 1999.

[5] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 90. ACM, 2013.

[6] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *CoRR*, abs/0709.1272, 2007.

[7] John Gunnels and Robert van de Geijn. Developing linear algebra algorithms: A collection of class projects. FLAME Working Note #3. Technical Report TR-2001-19, The University of Texas at Austin, Department of Computer Sciences, May 2001.

[8] Azzam Haidar, Hatem Ltaief, Asim YarKhan, and Jack Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency and Computation: Practice and Experience*, 24(3):305–321, 2012.

[9] Hatem Ltaief, Jakub Kurzak, and Jack Dongarra. Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):417–423, 2010.

[10] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):16, 2013.

[11] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.